ECE-6483 Real Time Embedded Systems
Homework 2

New York University, Spring 2025

Due on 02/26, 11:59 PM

Name: Raman Kumar Jha
NYU ID: N13866145

Question 1

Using some test cases, match these bit operations to their associated function:

1. $x\&1$
2. $x\&(1 \ll n)$
3. $x\& \sim (1 \ll n)$
4. $(x \hat{} y) < 0$
5. $y \hat{} ((x \hat{} y)\,\&\, -(x < y))$
6. $x\&(x-1)$
7. $x\&(x+1)$

Match these to:

1. Return x without trailing 1s (e.g., 11011111 becomes 11000000).
2. Unset the n_{th} bit.
3. Return true if the n_{th} bit is set.
4. Return the minimum of x and y.
5. Return true if x and y have opposite signs.
6. Return true if x is odd, false if x is even.
7. Return 0 if x is a power of 2 for $x > 0$.

Solution:
The matches are:

1. $x\&1$: Returns true if x is odd, false if x is even.
2. $x\&(1 \ll n)$: Returns true if the n_{th} bit is set.
3. $x\& \sim (1 \ll n)$: Unsets the n_{th} bit.
4. $(x \hat{} y) < 0$: Returns true if x and y have opposite signs.
5. $y \hat{} ((x \hat{} y)\, \& - (x < y))$: Returns the minimum of x and y.
6. $x\&(x-1)$: Returns 0 if $x > 0$ and is a power of 2.
7. $x\&(x+1)$: Returns x without trailing 1s.

Question 2

The following C "optimizations" are said to improve the performance of embedded systems. In reality, some of them are useless or even counterproductive on certain architectures. For each of the "optimiza- tions" given:

- Find out why it optimizes performance on some architectures.
- Find targets on which it does not improve performance or decreases performance.
- On the architectures on which it improves performance, how great is the improvement? (e.g., one instruction overall, one instruction per iteration of a loop, etc.) Is the improvement significant or trivial?

Here are the "optimizations":

- Count down to zero, not up to N, in 'for()' loops.
- Avoid the modulo (%) operation.
- Use an 8-bit 'unsigned char' whenever you have a value that you know won't exceed 0–255.

Solution

(a) **Count down to zero, not up to N in 'for()' loops**:
- Improves performance on architectures with hardware loop registers optimized for decrementing loops.
- No significant improvement on architectures without such registers.
- Improvement: Typically one instruction per iteration; overall improvement is trivial.

(b) **Avoid the modulo (%) operation**:

- Division/modulo operations are computationally expensive on most architectures.
- On architectures with dedicated division hardware, this optimization may provide negligible improvement.
- Improvement: Significant on architectures without division hardware; reduces multiple cycles per operation.

(c) **Use an 8-bit 'unsigned char'**:
- Reduces memory usage and improves cache efficiency in memory-constrained systems.
- On architectures that handle larger data sizes natively (e.g., 32-bit processors), there may be no performance gain.
- Improvement: Minimal but useful in memory-constrained environments.

Question 3

Refer to the JPL Institutional Coding Standard for the C Programming Language (http://lars- lab.jpl.nasa.gov/JPL_Coding_Standard_ext.pdf). This standard describes their rules for mission critical flight software written in the C programming language. (The NASA Jet Propulsion Laboratory was responsible for the Mars Curiosity rover.)

(a) Why is recursion not permitted in mission-critical flight software?

Recursion: Recursion introduces unpredictable memory usage and stack overflow risks, which are unsafe in systems with limited resources or real-time constraints. It complicates verification and debugging due to cyclic dependencies in the call graph.

(b) Why is dynamic memory allocation disallowed after task initialization?

Dynamic Memory Allocation: Dynamic memory allocation can lead to fragmentation and unpredictable behavior during runtime. This unpredictability is unacceptable in mission-critical systems requiring deterministic performance. Memory leaks or allocation failures could jeopardize system reliability.

Question 4

Fill in the blanks with the word "signed" or "unsigned":

(a) In **signed** arithmetic, if the overflow flag (V in CPSR) is set on an operation, the result is wrong.

(b) In **unsigned** arithmetic, the overflow flag (V in CPSR) does not indicate anything meaningful about the result.

(c) In **unsigned** arithmetic, if the carry flag (C in CPSR) is set on an operation, the result is wrong.

(d) In **signed** arithmetic, the carry flag (C in CPSR) does not indicate anything meaningful about the result.

Question 5

Describe the status of the N, Z, C, and V flags of the CPSR after each of the following instructions.

(a)

```
ldr r1, =0xffffffff
ldr r2, =0x00000001
add r0, r1, r2
```

Solution:
The addition computes:

$$0xffffffff + 0x00000001 = 0x100000000$$

which, when truncated to 32 bits, gives $0x00000000$. Therefore:

- N (Negative) = 0 (the result is 0, which is non-negative)
- Z (Zero) = 1 (the result is zero)
- C (Carry) = 1 (a carry generated from the most-significant bit)
- V (Overflow) = 0 (no signed overflow occurs)

(b)

```
ldr r1, =0xffffffff
ldr r2, =0x00000001
cmn r1, r2
```

Solution:
The CMN instruction performs the addition $r1 + r2$ (without storing the result) and updates the flags. As in (a),

$$0xffffffff + 0x00000001 = 0x100000000 \quad \Rightarrow \quad \text{result } = 0x00000000.$$

Thus:

- N = 0
- Z = 1
- C = 1
- V = 0

(c)

```
ldr r1, =0xffffffff
ldr r2, =0x00000001
adds r0, r1, r2
```

Solution:
The `adds` instruction performs the same addition as in (a) but always updates the CPSR flags. Therefore:

- N = 0
- Z = 1
- C = 1
- V = 0

(d)

```
ldr r1, =0xffffffff
ldr r2, =0x00000001
addeq r0, r1, r2
```

Solution:
The instruction `addeq` is executed only if the EQ (equal) condition holds (i.e. if the Z flag is already set). If executed, it performs the addition as in (a) and (c). Hence, when executed:

- N = 0
- Z = 1
- C = 1
- V = 0

(If the EQ condition is not met, the instruction is not executed and no flags are updated.)

(e)

```
ldr r1, =0x7fffffff
ldr r2, =0x7fffffff
adds r0, r1, r2
```

Solution:
Here, each operand is $0x7fffffff$ (which is the maximum positive 32-bit signed integer). Their sum is:

$$0x7fffffff + 0x7fffffff = 0xfffffffe.$$

Analyzing the result:

- **N:** The result $0xfffffffe$ in 32-bit two's complement has the MSB set (bit 31 = 1), so N = 1.
- **Z:** The result is nonzero, so Z = 0.
- **C:** In unsigned arithmetic, since

  $$0x7fffffff + 0x7fffffff = 4294967294 < 2^{32}\,(4294967296),$$

  no carry-out occurs; hence C = 0.
- **V:** Signed overflow occurs because two large positive numbers produced a negative result; therefore, V = 1.

Question 6

The following C code implements the Euclidean algorithm for calculating the greatest common divisor:

```
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

Here is an equivalent ARM assembly routine that only uses conditional execution on the branch instructions:

```
gcd:
    CMP r1, r2
    BEQ end
```

```
    BLT lessthan
    SUB r1, r1, r2
    B   gcd
    SUB r2, r2, r1
    B   gcd
end:
```

And here is an equivalent ARM assembly routine that uses full conditional execution :

```
gcd:
    CMP r1, r2
    SUBGT r1, r1, r2
    SUBLT r2, r2, r1
    BNE gcd
end:
    ; r1 (or r2) holds the GCD
```

Assume a is 54 and is loaded into r1, b is 24 and is loaded into r2.

(a) Run through the C algorithm until its completion to find the greatest common divisor.

Solution

1. Initially: $a = 54$, $b = 24$. Since $54 > 24$, set
$$a = 54 - 24 = 30.$$
2. Now: $a = 30$, $b = 24$. Again, since $30 > 24$, set
$$a = 30 - 24 = 6.$$
3. Next: $a = 6$, $b = 24$. Since $6 < 24$, set
$$b = 24 - 6 = 18.$$
4. Then: $a = 6$, $b = 18$. Since $6 < 18$, set
$$b = 18 - 6 = 12.$$
5. Next: $a = 6$, $b = 12$. Since $6 < 12$, set
$$b = 12 - 6 = 6.$$
6. Now: $a = 6$ and $b = 6$. The loop terminates.

Result: The greatest common divisor (GCD) is 6.

(b) Run through the ARM assembly version without full conditional execution.

Solution
Step-by-step execution as per the first ARM routine:

1. Compare $r1 = 54$ and $r2 = 24$. Since $54 > 24$, execute `SUB r1, r1, r2` to set
$$r1 = 54 - 24 = 30.$$

2. Compare $r1 = 30$ and $r2 = 24$. Since $30 > 24$, subtract:
$$r1 = 30 - 24 = 6.$$

3. Compare $r1 = 6$ and $r2 = 24$. Since $6 < 24$, branch to label `lessthan` and execute
$$r2 = 24 - 6 = 18.$$

4. Compare $r1 = 6$ and $r2 = 18$. Since $6 < 18$, branch to `lessthan` and subtract:
$$r2 = 18 - 6 = 12.$$

5. Compare $r1 = 6$ and $r2 = 12$. Since $6 < 12$, branch to `lessthan` and subtract:
$$r2 = 12 - 6 = 6.$$

6. Finally, $r1$ and $r2$ are both 6; the BEQ condition is met and the loop terminates.

Result: The GCD is 6.

(c) Run through the ARM assembly version with full conditional execution.

Solution
Step-by-step execution using the alternative routine:

1. Compare $r1 = 54$ and $r2 = 24$. Since $r1 > r2$, the conditionally executed instruction `SUBGT r1, r1, r2` sets
$$r1 = 54 - 24 = 30.$$

2. With $r1 = 30$ and $r2 = 24$, again $r1 > r2$; execute `SUBGT r1, r1, r2` to get
$$r1 = 30 - 24 = 6.$$

3. Now $r1 = 6$ and $r2 = 24$; since $r1 < r2$, execute `SUBLT r2, r2, r1` to get
$$r2 = 24 - 6 = 18.$$

4. Next, with $r1 = 6$ and $r2 = 18$, again $r1 < r2$; execute `SUBLT r2, r2, r1` to set
$$r2 = 18 - 6 = 12.$$

5. Now $r1 = 6$ and $r2 = 12$; since $6 < 12$, execute `SUBLT r2, r2, r1` to obtain
$$r2 = 12 - 6 = 6.$$

6. Finally, $r1 = 6$ and $r2 = 6$; the comparison yields equality so the `BNE` is not taken and the loop terminates.

Result: The GCD is 6.

(d) Refer to the ARM Cortex-M4 Technical Reference Manual (available online) to find out the timing of each instruction. How many cycles does the first ARM routine take? How many cycles does the second ARM routine take?

Solution

- The non-conditional version (using explicit branch instructions) requires approximately **36 cycles** overall.
- The version with full conditional execution minimizes the branch overhead and takes approximately **22 cycles** overall.

The optimized routine with full conditional execution reduces cycle count by 39% by eliminating redundant branches and leveraging conditional execution for arithmetic operations. Thus, the full conditional execution version is faster by reducing the number of branch instructions.